Using OpenBSD Security Features to Find Software Bugs

Peter Valchev
pvalchev@openbsd.org

Overview of OpenBSD

History

- Berkeley Software Distribution (BSD)
- First freely distributable BSD was released in June 1989.
- BSD License Origins
- GPL Comparison
- In 1993, NetBSD and FreeBSD were created.
- In 1995, OpenBSD was born.

Why OpenBSD?

- Project Culture
  - Small number of developers
  - Hackathons
  - Develop to help everyone, not just OpenBSD
- Pride ourselves on clean code and a friendly development environment.
- Open Source Advocacy
  - Use BSD license as much as possible
  - Audited licensing for the entire source tree.
  - Develop free device drivers (e.g. Atheros)
  - Fight for open hardware documentation.

OpenBSD Security Goals

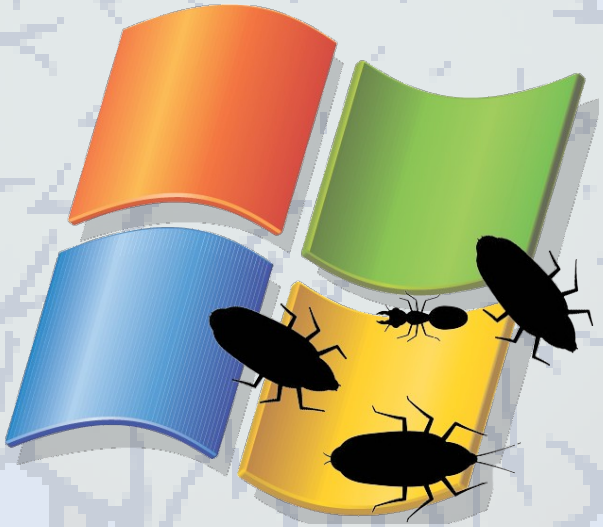
- OpenBSD has an uncompromising view toward security.
- “Secure by Default”
- Only 2 remote holes in the default install, in more than 10 years! (How many has Windows had?)
- Use and design cutting edge security technologies.
- Strong use of cryptography and hardware crypto.
- Continual auditing
- More at: http://www.openbsd.org/security.html

The Main Idea

- OpenBSD can be considered a hostile environment.
- This environment allows us to find bugs in applications that would otherwise silently misbehave.
- We are constrained by POSIX and ANSI, but still have lots of room to play.
- As a result of our work, we have not only improved the base system, but have found many issues in third party packages such as Firefox, OpenOffice, etc.

Security and Bugs

- Who has had their box 0wned? Who has 0wned a box?
- A security bug is just a bug.
- Understanding and reproducing its side effects creates an opportunity for system exploit.

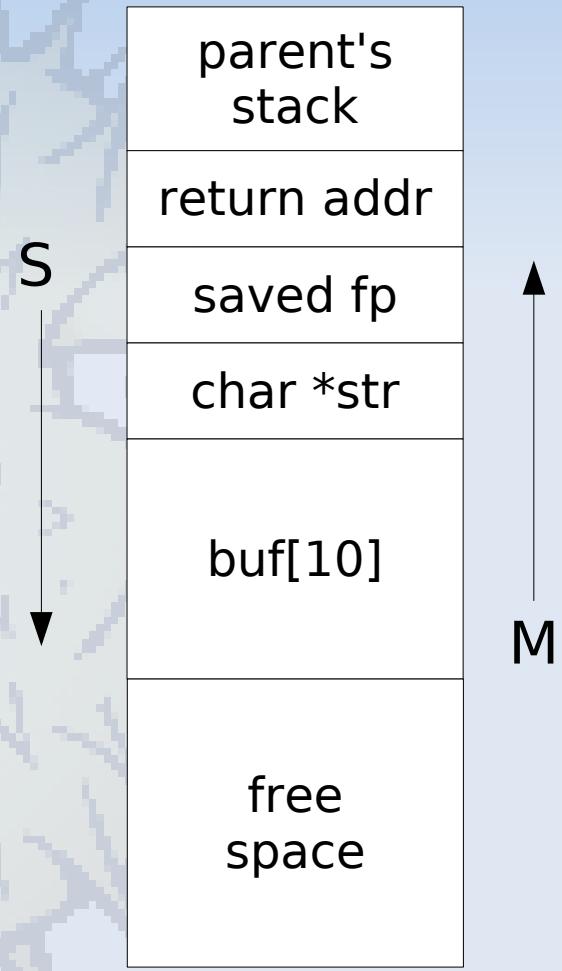


Some Security Technologies

- strlcpy/strlcat
- memory protection
- privilege revocation (e.g. ping)
- privilege separation (e.g. SSH)
- chroot jailing
- new uids
- ProPolice (SSP)
- StackGap (3-line change to the kernel)
- ld.so randomness
- mmap and malloc randomness

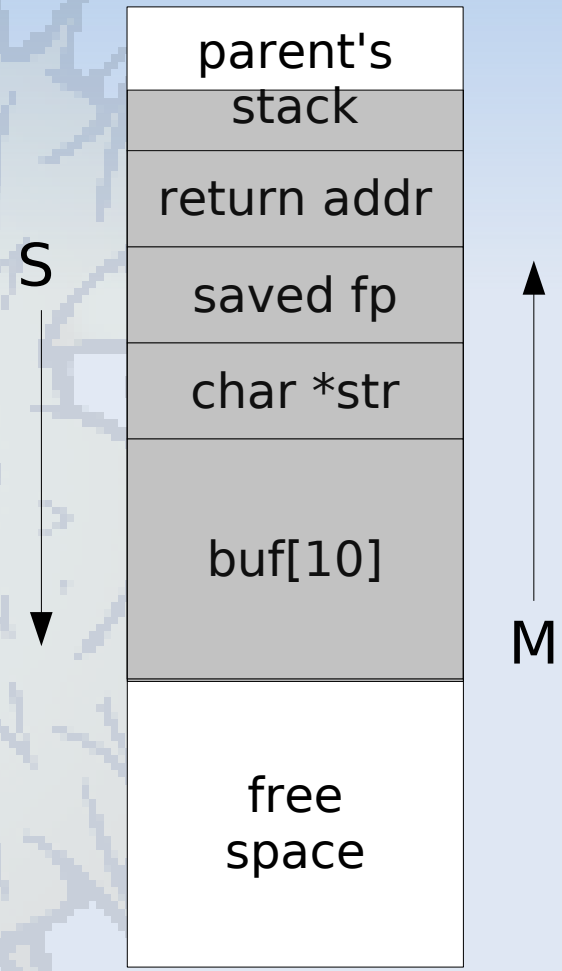
Stack Buffer Overflows

```
void foo(char *str) {
     char buf[10];
     strcpy(buf, str);
}
```



Stack Buffer Overflows

```
void foo(char *str) {
     char buf[10];
     strcpy(buf, str);
}
```

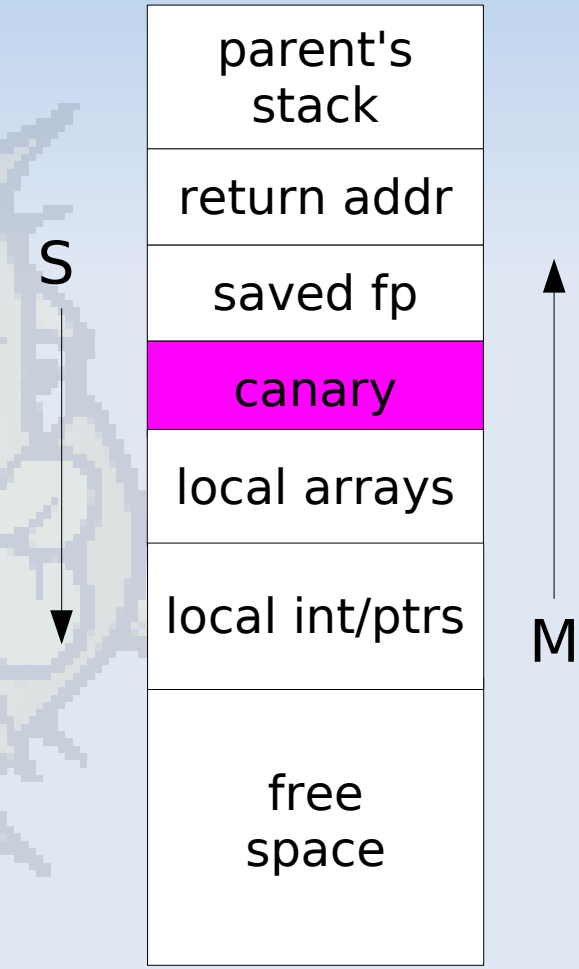


ProPolice (SSP)

- A compiler extension for protecting against stack smashing attacks.
- Protection code is inserted into the programs at compile time.
- Should be default (negligible overhead), but can also be enabled with a flag.
- Runtime protection “after the fact”
- Default in OpenBSD since December 2002; recently in gcc-4, but not default!

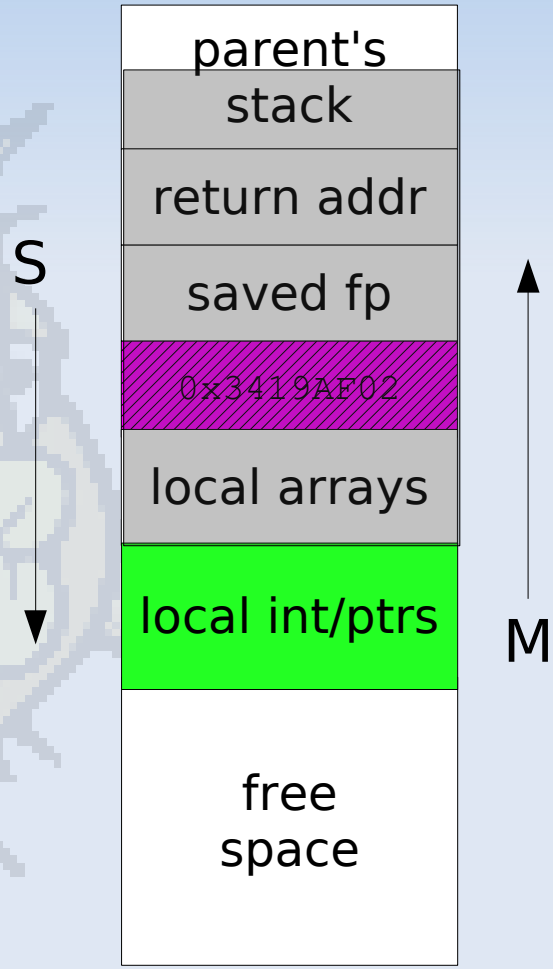
ProPolice, How It Works

- Protection code is injected into each candidate function at compile time.
  - Prologue inserts a random value, the ***canary***, on the stack before local arrays
  - Function Epilogue checks the canary and aborts if it has been modified
- Reorder the stack by putting buffers closer to the return address.
  - Flags and pointers are lower, so they are harder to overwrite.
  - Overflows are more likely to modify the canary.
- Low cost at compile time; performance impact of ~1.3%



ProPolice, How It Works

- Protection code is injected into each candidate function at compile time.
  - Prologue inserts a random value, the ***canary***, on the stack before local arrays
  - Function Epilogue checks the canary and aborts if it has been modified
- Reorder the stack by putting buffers closer to the return address.
  - Flags and pointers are lower, so they are harder to overwrite.
  - Overflows are more likely to modify the canary.
- Low cost at compile time; performance impact of ~1.3%



ProPolice Demo

Step 1. Compile program with ProPolice-enabled GCC

Step 2. Done! Program can be ran as normal.

If an overflow is detected upon execution, the program is terminated and a message is logged.

Spot the Bug!

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#define MAXLEN 1024


int main() {
    char *string, *test;
    string = strdup("Hey World");
    test = malloc(MAXLEN);
    strlcpy(test, string, sizeof(string));
    printf("%s\n", test);
    return 1;
}
```

gcc -Wbounded

- A compiler option (gcc -Wbounded) that performs static checking to make sure the bounds length passed to common functions matches the real buffer length.

```
char buf[1024]           // buf's size has to be known at compile time
snprintf(buf, SIZE, fmt) // will check if SIZE == 1024
```

- -Wformat also enables bounds checking for scanf(3) %s format variables.
- Right now this checking is very limited and simple, and still found **hundreds of bugs** throughout our ports tree (in **third party applications**).

DEMO!

Randomization Efforts

- Exploits rely on predictable system behaviour. How can we cause exploit writers hell?
- StackGap inserts a random-sized gap in the stack.
- randomization in ld.so(1)
  - Load libraries at random memory locations
  - Load libraries in random order
- random mmap(2)
- random malloc(3)

random mmap(2)

- POSIX says that if **MAP_FIXED** is specified, we must map into that address.
- Otherwise, the OS is free to select a memory location (other systems do this very predictably).
- So, if MAP_FIXED is not specified, choose a **random** address. Each time a program is ran, different address space behaviour!
- When relying on finding a function/code at a certain location, must guess where it's placed in virtual memory.

random malloc(3)

- The addresses of objects allocated by malloc() are quite predictable!
- Exploits have relied on this!
- Two types of objects
  - < PAGE_SIZE: malloc maintains a bucket of “chunks” and returns a random one; this is not default behaviour yet.
  - ≥ PAGE_SIZE: use mmap(2), which is randomized

DEMO!

malloc “Guard Pages”

- Enabled with /etc/malloc.conf -> 'G'
- Enables guard pages and chunk randomization. Each page size or larger allocation is followed by a guard page that will cause a segmentation fault upon access. Smaller than page size chunks are returned in a random order.
- Other robust measures (eg. malloc(0) crashes)
- Not default, since too many applications are buggy

DEMO!

Address Space Policy: W^X

- Many exploits rely on the fact that the address space has memory that is both writeable and executable (W | X)
- Make a generic policy for the whole address space:
- A page may be writeable or executable, but not both (W^X)
  - Need per-page X bit (fine-grained page permissions): arm, amd64, sparc64, many others
  - Other solution necessary for i386/powerpc
- Found several bugs in applications that made incorrect assumptions. We still conform to POSIX, ANSI, etc.

Architecture Differences

- OpenBSD supports 12 different architectures (17 platforms). Why is this cool?
- Helps us uncover machine-independent bugs!
- Examples
  - big endian vs little endian (eg. sparc64 vs i386)
  - 64-bit vs 32-bit
  - signed vs unsigned char (macppc)
  - stack grows **up** not down (where?)
  - aligned memory access requirement
  - ILP32 vs I32LP64 (nastiest is sparc64 - BE I32LP64!)

Conclusion

Features we discussed

- Compiler Features (ProPolice, Wbounded)
- Memory Protection (non-exec stack, W^X, others)
- Randomization (stackgap, ld.so, mmap, malloc)
- Malloc Guard Pages

Using these, we have uncovered a lot of evil! We have created a hostile environment for applications to run in, and that has provided safer, more robust software!

Thanks

- Thanks for having me here!
- Thanks to the other OpenBSD developers that built all of this!
- The project is supported by donations and sales of CDs and T-Shirts (I have some here!)

References:

- http://www.openbsd.org/papers/strlcpy-paper.ps
- http://www.trl.ibm.com/projects/security/ssp

Remember...


Every time you use Flash, God kills a kitten.
Please think of the kittens.